The ryg blog
When I grow up I'll be an inventor.

A trip through the Graphics Pipeline 2011, part 13

October 9, 2011
*This post is part of the series **"A trip through the Graphics Pipeline 2011" (https://fgiesen.wordpress.com/2011/07/09/a-trip-through-the-graphics-pipeline-2011-index/)**.*

Welcome back to what's going to be the last "official" part of this series – I'll do more GPU-related posts in the future, but this series is long enough already. We've been touring all the regular parts of the graphics pipeline, down to different levels of detail. Which leaves one major new feature introduced in DX11 out: Compute Shaders. So that's gonna be my topic this time around.

Execution environment

For this series, the emphasis has been on overall dataflow at the architectural level, not shader execution (which is explained well elsewhere). For the stages so far, that meant focusing on the input piped into and output produced by each stage; the way the internals work was usually dictated by the shape of the data. Compute shaders are different – they're running by themselves, not as part of the graphics pipeline, so the surface area of their interface is much smaller.

In fact, on the input side, there's not really any buffers for input data at all. The only input Compute Shaders get, aside from API state such as the bound Constant Buffers and resources, is their thread index. There's a tremendous potential for confusion here, so here's the most important thing to keep in mind: a "thread" is the atomic unit of dispatch in the CS environment, and it's a substantially different beast from the threads provided by the OS that you probably associate with the term. CS threads have their own identity and registers, but they don't have their own Program Counter (Instruction Pointer) or stack, nor are they scheduled individually.

In fact, "threads" in CS take the place that individual vertices had during **Vertex Shading (https://fgiesen.wordpress.com/2011/07/03/a-trip-through-the-graphics-pipeline-2011-part-3/)**, or individual pixels during **Pixel Shading (https://fgiesen.wordpress.com/2011/07/10/a-trip-through-the-graphics-pipeline-2011-part-8/)**. And they get treated the same way: assemble a bunch of them (usually, somewhere between 16 and 64) into a "Warp" or "Wavefront" and let them run the same code in lockstep. CS threads don't get scheduled – Warps and Wavefronts do (I'll stick with "Warp" for the rest of this article; mentally substitute "Wavefront" for AMD). To hide latency, we don't switch to a different "thread" (in CS parlance), but to a different Warp, i.e. a different bundle of threads. Single threads inside a Warp can't take branches individually; if at least one thread in such a bundle wants to execute a certain piece of code, it gets processed by all the threads in the bundle – even if most threads then end up throwing the results away. In short, CS "threads" are more like SIMD lanes than like the threads you see elsewhere in programming; keep that in mind.

That explains the "thread" and "warp" levels. Above that is the "thread group" level, which deals with – who would've thought? – groups of threads. The size of a thread group is specified during shader compilation. In DX11, a thread group can contain anywhere between 1 and 1024 threads, and the thread group size is specified not as a single number but as a 3-tuple giving thread x, y, and z coordinates. This numbering scheme is mostly for the convenience of shader code that addresses 2D or 3D resources, though it also allows for traversal optimizations. At the macro level, CS execution is dispatched in multiples of thread groups; thread group IDs in D3D11 again use 3D group IDs, same as thread IDs, and for pretty much the same reasons.

Thread IDs – which can be passed in in various forms, depending on what the shader prefers – are the only input to Compute Shaders that's not the same for all threads; quite different from the other shader types we've seen before. This is just the tip of the iceberg, though.

Thread Groups

The above description makes it sound like thread groups are a fairly arbitrary middle level in this hierarchy. However, there's one important bit missing that makes thread groups very special indeed: Thread Group Shared Memory (TGSM). On DX11 level hardware, compute shaders have access to 32k of TGSM, which is basically a scratchpad for communication between threads in the same group. This is the primary (and fastest) way by which different CS threads can communicate.

So how is this implemented in hardware? It's quite simple: all threads (well, Warps really) within a thread group get executed by the same shader unit. The shader unit then simply has at least 32k (usually a bit more) of local memory. And because all grouped threads share the same shader unit (and hence the same set of ALUs etc.), there's no need to include complicated arbitration or synchronization mechanisms for shared memory access: only one Warp can access memory in any given cycle, because only one Warp gets to issue instructions in any cycle! Now, of course this process will usually be pipelined, but that doesn't change the basic invariant: per shader unit, we have exactly one piece of TGSM; accessing TGSM might require multiple pipeline stages, but actual reads from (or writes to) TGSM will only happen inside one pipeline stage, and the memory accesses during that cycle all come from within the same Warp.

However, this is not yet enough for actual shared-memory communication. The problem is simple: The above invariant guarantees that there's only one set of accesses to TGSM per cycle even when we don't add any interlocks to prevent concurrent access. This is nice since it makes the hardware simpler and faster. It does not guarantee that memory accesses happen in any particular order from the perspective of the shader program, however, since Warps can be scheduled more or less randomly; it all depends on who is runnable (not waiting for memory access / texture read completion) at certain points in time. Somewhat more subtle, precisely because the whole process is pipelined, it might take some cycles for writes to TGSM to become "visible" to reads; this happens when the actual read and write operations to TGSM occur in different pipeline stages (or different phases of the same stage). So we still need some kind of synchronization mechanism. Enter barriers. There's different types of barriers, but they're composed of just three fundamental components:

1. *Group Synchronization*. A Group Synchronization Barrier forces all threads inside the current group to reach the barrier before any of them may consume past it. Once a Warp reaches such a barrier, it will be flagged as non-runnable, same as if it was waiting for a memory or texture access to complete. Once the last Warp reaches the barrier, the remaining Warps will be reactivated. This all happens at the Warp scheduling level; it adds additional scheduling constraints, which may cause stalls, but there's no need for atomic memory transactions or anything like that; other than lost utilization at the micro level, this is a reasonably cheap operation.
2. *Group Memory Barriers*. Since all threads within a group run on the same shader unit, this basically amounts to a pipeline flush, to ensure that all pending shared memory operations are completed. There's no need to synchronize with resources external to the current shader unit, which means it's again reasonably cheap.
3. *Device Memory Barriers*. This blocks all threads within a group until all memory accesses have completed – either direct or indirect (e.g. via texture samples). As explained earlier in this series, memory accesses and texture samples on GPUs have long latencies – think more than 600, and often above 1000 cycles – so this kind of barrier will really hurt.

DX11 offers different types of barriers that combine several of the above components into one atomic unit; the semantics should be obvious.

Unordered Access Views

We've now dealt with CS input and learned a bit about CS execution. But where do we put our output data? The answer has the unwieldy name "unordered access views", or UAVs for short. An UAV seems somewhat similar to render targets in Pixel Shaders (and UAVs can in fact be used in addition to render targets in Pixel Shaders), but there's some very important semantic differences:

> Most importantly, as the same suggests, access to UAVs is "unordered", in the sense that the API does not guarantee accesses to become visible in any particular order. When rendering primitives, quads are guaranteed to be Z-tested, blended and written back in API order (as discussed in detail in **part 9 of this series (https://fgiesen.wordpress.com/2011/07/12/a-trip-through-the-graphics-pipeline-2011-part-9/)**), or at least produce the same results as if they were – which takes substantial effort. UAVs make no such effort – UAV accesses happen immediately as they're encountered in the shader, which may be very different from API order. They're not *completely* unordered, though; while there's no guaranteed order of operations within an API call, the API and driver will still collaborate to make sure that perceived sequential ordering is preserved across API calls. Thus, if

you have a complex Compute Shader (or Pixel Shader) writing to an UAV immediately followed by a second (simpler) CS that reads from the same underlying resource, the second CS will see the finished results, never some partially-written output.
UAVs support random access. A Pixel Shader can only write to one location per render target – its corresponding pixel. The same Pixel Shader can write to arbitrary locations in whatever UAVs it has bound.
UAVs support atomic operations. In the classic Pixel Pipeline, there's no need; we guarantee there's never any collisions anyway. But with the free-form execution provided by UAVs, different threads might be trying to access a piece of memory at the same time, and we need synchronization mechanisms to deal with this.

So from a "CPU programmer"'s point of view, UAVs correspond to regular RAM in a shared-memory multiprocessing system; they're windows into memory. More interesting is the issue of atomic operations; this is one area where current GPUs diverge considerably from CPU designs.

Atomics

In current CPUs, most of the magic for shared memory processing is handled by the memory hierarchy (i.e. caches). To write to a piece of memory, the active core must first assert exclusive ownership of the corresponding cache line. This is accomplished using what's called a "cache coherency protocol", usually **MESI (http://en.wikipedia.org/wiki/MESI_protocol)** and descendants. The details are tangential to this article; what matters is that because writing to memory entails acquiring exclusive ownership, there's never a risk of two cores simultaneously trying to write to the some location. In such a model, atomic operations can be implemented by holding exclusive ownership for the duration of the operation; if we had exclusive ownership for the whole time, there's no chance that someone else was trying to write to the same location while we were performing the atomic operation. Again, the actual details of this get hairy pretty fast (especially as soon as things like paging, interrupts and exceptions get involved), but the 30000-feet-view will suffice for the purposes of this article.

In this type of model, atomic operations are performed using the regular Core ALUs and load/store units, and most of the "interesting" work happens in the caches. The advantage is that atomic operations are (more or less) regular memory accesses, albeit with some extra requirements. There's a couple of problems, though: most importantly, the standard implementation of cache coherency, "snooping", requires that all agents in the protocol talk to each other, which has serious scalability issues. There are ways around this restriction (mainly using so-called Directory-based Coherency protocols), but they add additional complexity and latency to memory accesses. Another issue is that all locks and memory transactions really happen at the cache line level; if two unrelated but frequently-updated variables share the same cache line, it can end up "ping-ponging" between multiple cores, causing tons of coherency transactions (and associated slowdown). This problem is called "false sharing". Software can avoid it by making sure unrelated fields don't fall into the same cache line; but on GPUs, neither the cache line size nor the memory layout during execution is known or controlled by the application, so this problem would be more serious.

Current GPUs avoid this problem by structuring their memory hierarchy differently. Instead of handling atomic operations inside the shader units (which again raises the "who owns which memory" issue), there's dedicated atomic units that directly talk to a shared lowest-level cache hierarchy. There's only one such cache, so the issue of coherency doesn't come up; either the cache line is present in the cache (which means it's current) or it isn't (which means the copy in memory is current). Atomic operations consist of first bringing the respective memory location into the cache (if it isn't there already), then performing the required read-modify-write operation directly on the cache contents using a dedicated integer ALU on the atomic units. While an atomic unit is busy on a memory location, all other accesses to that location will stall. Since there's multiple atomic units, it's necessary to make sure they never try to access the same memory location at the same time; one easy way to accomplish this is to make each atomic unit "own" a certain set of addresses (statically – not dynamically as with cache line ownership). This is done by computing the index of the responsible atomic unit as some hash function of the memory address to be accessed. (Note that I can't confirm this is how current GPUs do; I've found little detail on how the atomic units work in official docs).

If a shader unit wants to perform an atomic operation to a given memory address, it first needs to determine which atomic unit is responsible, wait until it is ready to accept new commands, and then submit the operation (and potentially wait until it is finished if the result of the atomic operation is required). The atomic unit might only be processing one command at a time, or it might have a small FIFO of outstanding requests; and of course there's all kinds of allocation and queuing details to get right so that atomic operation processing is reasonably fair so that shader units will always make progress. Again, I won't go into further detail here.

One final remark is that, of course, outstanding atomic operations count as "device memory" accesses, same as memory/texture reads and UAV writes; shader units need to keep track of their outstanding atomic operations and make sure they're finished when they hit device memory access barriers.

Structured buffers and append/consume buffers

Unless I missed something, these two buffer types are the last CS-related features I haven't talked about yet. And, well, from a hardware perspective, there's not that much to talk about, really. Structured buffers are more of a hint to the driver-internal shader compiler than anything else; they give the driver some hint as to how they're going to be used – namely, they consist of elements with a fixed stride that are likely going to be accessed together – but they still compile down to regular memory accesses in the end. The structured buffer part may bias the driver's decision of their position and layout in memory, but it does not add any fundamentally new functionality to the model.

Append/consume buffers are similar; they could be implemented using the existing atomic instructions. In fact, they kind of are, except the append/consume pointers aren't at an explicit location in the resource, they're side-band data outside the resource that are accessed using special atomic instructions. (And similarly to structured buffers, the fact that their usage is declared as append/consume buffer allows the driver to pick their location in memory appropriately).

Wrap-up.

And… that's it. No more previews for the next part, this series is done :), though that doesn't mean I'm done with it. I have some restructuring and partial rewriting to do – these blog posts are raw and unproofed, and I intend to go over them and turn it into a single document. In the meantime, I'll be writing about other stuff here. I'll try to incorporate the feedback I got so far – if there's any other questions, corrections or comments, now's the time to tell me! I don't want to nail down the ETA for the final cleaned-up version of this series, but I'll try to get it down well before the end of the year. We'll see. Until then, thanks for reading!

From → Coding, Graphics Pipeline

14 Comments

1. **Joerg permalink**
   Very fine granular (per-thread) scheduling is briefly mentioned for the T600 http://blogs.arm.com/multimedia/534-memory-management-on-embedded-graphics-processors.

   Reply
2. **Alex permalink**
   Thanks for a fantastic series .. some of the best technical writing around, for its depth, clarity and content.

   Reply
3. **wuyuwen permalink**
   Terrific series, Thanks!

   Reply
4. **luke permalink**
   Thanks for this series. It was a pleasure to read.

   Reply
5. **Marcel Ancel permalink**
   Hi. Very interesting articles. Thanks a lot
   and thanks for your demos, great job :)

   Reply
6. **jt permalink**
   Thanks for the series. Do you have a higher level document for the not-so-advanced audience? How does the game content (models, textures) get into the game level? When we launch a game's exe, what goes on in the background? Where do the DX APIs and gpu drivers come into picture when running a game?
   I've looked all over the internet but haven't found anything that describes these in an easy to understand way.

   Reply
7. **yoelshoshan permalink**
   Thanks a lot for this great series!
   I found it very useful :)

   Reply

8. **Augus1990 permalink**
   Fantastic, thank you for all this info. It would be great if you make a PDF with all the information that you write about Graphics Pipeline, but it's just a suggestion.

   I sorry for my english, bye.

   Reply
9. **royalestel permalink**
   Holy crap. I thought I was somewhat knowledgeable about computer graphics, since It's been my hobby for a long while now. I see now that I am nowhere close! Thanks for this great series! Cheers!

   Reply
10. **Tommy permalink**
    This stuff is gold!

    Hope for dx12 vulkan update!

    Reply
11. **Martijn permalink**
    A very interesting series. Even though it's currently jan. 2016 it was still very useful for me to read and I learned a lot from it.

    I have one question (though I understand if you're done with this series by now, it's been 5 years after all…) about thread groups:

    Do I understand correctly that one *and only one* thread group will run on a shader unit at the same time? Because different thread groups would mean they would stomp on each others TGSM I'd think. That would mean you should never make your thread groups too small, or you will get underutilisation of your shader units because of waiting for memory latency, but also not too big or you'll force too many warps onto a single shader unit (and possibly leave whole units unused).
    Do I understand this correctly?

    I assume this is the same as the local workgroup size in OpenCL, where the documentation gives some vague handwaving like 'the ideal local workgroup size depends on your problem and the hardware' bot no actual insights in how to decide a good goup size.

    If reading this series gave me the correct insight in this particular point, only that would have been an afternoon well spent (but I learned tons of things from the rest as well).

    Reply
    - **fgiesen permalink**
      I would've updated it, but GPU architecture (or at least the parts I'm describing) hasn't substantially changed in the last 5 years at all. Microarchitecture yes (the usual incremental improvements), but the big picture has been pretty stable.

      Anyway, "shader units" (reading this to mean EUs for Intel GPUs, CUs for AMD GCN and SMXs for NVidia GPUs) can run multiple thread groups at the same time no problem. Thread group memory is allocated in a similar way to GPU registers: the hardware units have a fixed-size pool, and the maximum number of active thread groups on a shader unit depends (among other things) on how many of them fit in said pool at the same time. (Subject to a certain allocation granularity which depends on the hardware, and also subject to other restrictions that again depend on the hardware).

      This series purposefully stays away from shader core internals since that was very much in flux at the time of writing, and besides, it's the part that the vendors tend to tell you lots of details about anyway.

      In general, you want to keep your thread groups relatively small if possible. Around 64 threads is a good value for current AMD and NV hardware (Intel prefers less). AMD GCN hardware cannot run less than 64 threads (1 wavefront) in a thread group; make them any smaller and they'll still allocate 64 threads worth of resources to it. Current NV hardware has a similar limit (warp size), though theirs is 32 threads (though 64 is still often better). Other than that, a thread group is a unit of threads that can communicate efficiently. Depending on the task, sometimes you want more. If you don't care much, staying close to 64 is a good rule of thumb at the moment.

      Reply
12. **Chris permalink**
    It tooks me three hours to find I was looking for. Thank you very much for this interesting and profound article about GPU parallelity!

    Reply

Trackbacks & Pingbacks

1. A trip through the Graphics Pipeline 2011: Index « The ryg blog